Verifying Binarized Neural Networks by Local Automaton Learning

Andy Shih and **Adnan Darwiche** and **Arthur Choi**
Computer Science Department
University of California, Los Angeles
{andyshih,darwiche,aychoi}@cs.ucla.edu

Abstract

We consider the problem of verifying the behavior of a Binarized Neural Network (BNN) on some input region. We propose an algorithm that compiles the BNN on the given region into an Ordered Binary Decision Diagram (OBDD), allowing us to efficiently answer a range of verification queries on the compiled OBDD. This includes counting and computing the probability of counterexamples, in addition to identifying the characteristics of counterexamples. Our compilation algorithm is based on a classical algorithm for learning Deterministic Finite Automaton (DFA) and some of its variations. We also present some preliminary experimental results based on BNNs for recognizing digits in images.

1 Introduction

Neural networks are used for a wide array of tasks, including speech recognition, image classification, and language translation. They also power safety-critical applications, such as autonomous driving, where humans need to understand and formally verify the behavior of underlying neural networks. While recent advancements have improved the performance and scale of neural networks, there has been a lack of enough methods for providing formal guarantees about their behavior. In addition, the complex and intricate structure of neural networks makes it impractical to reason about their behavior manually. This has sparked a recent line of research that aims to automatically verify neural network properties.

We propose in this paper an approach for verifying the properties of neural networks, which is based on knowledge compilation (Selman and Kautz, 1996; Cadoli and Donini, 1997; Darwiche and Marquis, 2002; Darwiche, 2014). We focus on the class of Binarized Neural Networks (BNN) (Hubara et al., 2016), which have binary weights and activations at runtime, leading to space and computational efficiencies. BNNs have also been shown to achieve comparable performance to traditional floating point precision networks on some standard datasets (Hubara et al., 2016).

One particular property of BNNs that has been studied is robustness (Leofante et al., 2018). Users of a BNN can pinpoint a particular input instance $\mathbf{x}$ and ask for guarantees on the behavior of the BNN for other inputs in the neighborhood of $\mathbf{x}$, which we denote by $S_{\mathbf{x}}$ and call an *input region.* This has practical applications, e.g., for image classification, where users expect an image of, say, a dog to remain classified as a dog if only a few pixels are modified. Since the number of ways to tweak an image is exponential in the number of modified pixels, it is impractical to perform the verification by enumeration.

A method was recently proposed for detecting counterexamples in an input region $S_{\mathbf{x}}$ (Narodytska et al., 2018). Our proposed approach pushes this direction further by harnessing techniques from knowledge compilation, allowing one to also *reason* about counterexamples. For example, we can efficiently count the counterexamples in $S_{\mathbf{x}}$, compute their probability, enumerate a subset of them, and identify their common characteristics. Another useful query supported by our approach, the *prime-implicant query,* returns a subset of inputs that, if fixed, will guarantee that the neural network output will stick even if we vary the unfixed inputs (Shih, Choi, and Darwiche, 2018a).

Using the example of image classification, our new techniques allow us to perform reasoning on all images that are some pixels away from some target image I, say, of a dog. Whereas previous methods only tell us that it is possible to classify an image in the neighborhood of I as, say, a cat, we can determine how many neighborhood images are classified as cats and identify key characteristics that are shared among all such images. The prime-implicant query even identifies a way to fix a minimal set of pixels in the dog image that guarantees a correct classification even if we modify some of the unfixed pixels.

To reason about BNNs, we compile them into a tractable representation and then apply verification queries to the compiled representation. The compilation is done once per input region and, if successful, allows one to efficiently answer a range of queries that are generally NP-hard (Shih, Choi, and Darwiche, 2018a).

We now give an overview of our compilation algorithm. We compile BNNs into the tractable Ordered Binary Decision Diagrams (OBDDs), which are decision graphs with an enforced variable ordering (Bryant, 1986; Meinel and Theobald, 1998; Wegener, 2000). Let B be a BNN, and let B_S represent the function of B on S, an input region of interest. To obtain B_S in a tractable form, we provide a novel algorithm for learning an OBDD representation of B_S, using a variation of the automaton learning algorithm from (Angluin, 1987). This variation learns the OBDD using standard membership and equivalence queries (Nakamura, 2005). Our algorithm constructs a hypothesis OBDD and then iteratively calls equivalence queries, adding OBDD nodes until its output agrees with B_S. To answer equivalence queries efficiently, we encode the BNN and the hypothesis OBDD into CNF, and require that the region S can be encoded as a CNF as well. When the algorithm terminates, it returns an OBDD D such that $D(\mathbf{x}) = B(\mathbf{x}) : \forall \mathbf{x} \in S$, a notion related to the `Constrain` operator on OBDDs (Meinel and Theobald, 1998). We then verify properties of BNN B by performing efficient verification queries on OBDD D.

Our algorithm can also be used as an incremental and anytime compilation algorithm, by slowly increasing the region of interest. The compiled OBDD of a smaller region can be used as the hypothesis OBDD for the compilation task of a larger region, without starting over. We can essentially save our progress, and build on it at a later time if we decide the initial region is too small.

This paper is structured as follows. Section 2 provides an introduction to BNNs and OBDDs. Section 3 describes the encodings of BNNs and OBDDs into CNF. Section 4 goes over the variation of the automaton learning algorithm, which is used by our compilation algorithm in Section 5. Preliminary experimental results relating to scalability are then given in Section 6, followed by a case study in Section 7. We finally discuss related work in Section 8 and conclude in Section 9.

2 Background

In this section, we describe Binarized Neural Networks and Ordered Binary Decision Diagrams in more detail.

Binarized Neural Networks

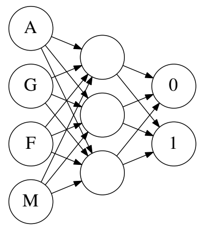
A Binarized Neural Network is a feed-forward neural network where the weights and activations are binarized using $\{-1, 1\}$. A BNN is composed of internal blocks and one output block. Internal blocks consist of three layers: a linear transformation (LIN), batch normalization (BN), and binarization (BIN).

- The LIN layer has parameters $\mathbf{a}$ (weights) and b (bias). Given an input $\mathbf{x}$, this layer returns $\langle \mathbf{a}, \mathbf{x} \rangle + b$.
- The BN layer has parameters μ (mean), σ (standard deviation), α (weight), and γ (bias). Given an input y, this layer returns $\alpha(\frac{y-\mu}{\sigma}) + \gamma$.
- The BIN layer returns the sign (1 or -1) of its input.

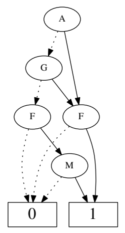
The output block consists of a LIN layer and an ARGMAX layer. The ARGMAX layer picks the output class with the highest activation. More details regarding these blocks and layers and their exact definitions can be found in (Narodytska et al., 2018). For convenience we consider a BNN with two output classes 0 and 1.

Ordered Binary Decision Diagrams

An *Ordered Binary Decision Diagram* (OBDD) is a *tractable* representation of a Boolean function over variables $\mathbf{X} = X_1, \ldots, X_n$ (Bryant, 1986; Meinel and Theobald, 1998; Wegener, 2000). An OBDD is a rooted, directed



(a) BNN with two outputs $\{0, 1\}$. The output with higher activation is the classification.



(b) OBDD with sinks $\{0, 1\}$.

Figure 1: A BNN and its corresponding OBDD on four inputs. The two representations compute the same function.

acyclic graph with two sinks called the 1-sink and 0-sink. Every node (except the sinks) in the OBDD is labeled with a variable X_i and has two labeled outgoing edges: the 1-edge and the 0-edge. The labeling of the OBDD nodes respects some global ordering of the variables $\mathbf{X}$: if there is an edge from a node labeled X_i to a node labeled X_j, then X_i must come before X_j in the global ordering. To evaluate the OBDD on an instance $\mathbf{x}$, start at the root node of the OBDD. Let x_i be the value of variable X_i of the current node. Repeatedly follow the x_i-edge of the current node, until a sink node is reached. Reaching the 1-sink means $\mathbf{x}$ is evaluated to 1 and reaching the 0-sink means $\mathbf{x}$ is evaluated to 0 by the OBDD. Hence, an OBDD can be viewed as representing a function $f(\mathbf{X})$ that maps instances $\mathbf{x}$ into $\{0, 1\}$.

Consider the BNN in Figure 1a, which classifies a movie as a box-office success or not. It has four binary inputs: A (*Adapted Screenplay*), G (*Great Cinematography*), F (*Famous Cast*), and M (*Marketing*). The parameters of the BNN are not shown, but it computes the truth table as shown in Table 1. The OBDD in Figure 1b also computes the truth table in Table 1, so we can verify properties of the BNN by performing verification on the OBDD. We can examine, for example, a movie that is an adapted screenplay, has great cinematography, a famous cast, heavy marketing, and is classified as being a box office success. This movie corresponds to input {A=1,G=1,F=1,M=1} and a classification of 1. Using the OBDD in Figure 1b we can deduce, in time linear on the size of the OBDD, that the movie could have had poor cinematography and low marketing, and would still be classified as being a box office success. In fact, the partial input {A=1,F=1} completely determines that the movie will be classified as being successful, regardless of how the remaining input is set. This is an example of the many types of efficient verification queries that can be done on an OBDD.

3 CNF Encodings

We next provide the encoding of BNNs and OBDDs into CNF, which will serve an important role in our main compilation algorithm.

	A	G	F	M	$f(\mathbf{x})$
1	-	-	-	-	-
2	-	-	-	+	-
3	-	-	+	-	-
4	-	-	+	+	+
5	-	+	-	-	-
6	-	+	-	+	-
7	-	+	+	-	+
8	-	+	+	+	+
9	+	-	-	-	-
10	+	-	-	+	-
11	+	-	+	-	+
12	+	-	+	+	+
13	+	+	-	-	-
14	+	+	-	+	-
15	+	+	+	-	+
16	+	+	+	+	+

Table 1: The Boolean function on the 16 possible inputs computed by the BNN and OBDD in Figure 1.

BNN to CNF

We use the conversion from (Narodytska et al., 2018). An internal block of a BNN consists of three layers: a linear transformation (LIN), batch normalization (BN), and binarization (BIN). The LIN layer has parameters $\mathbf{a}$ (weights) and b (bias). The BN layer has parameters μ (mean), σ (std), α (weight), and γ (bias). Put together, the three layers of an internal block can be translated to the following function $h(\mathbf{x})$ on an input instance $\mathbf{x}$ (Narodytska et al., 2018).

$$h(\mathbf{x}) = 1 \iff \langle \mathbf{a}, \mathbf{x} \rangle \geq -\frac{\sigma}{\alpha}\gamma + \mu - b$$

Since the weights $\mathbf{a}$ and input $\mathbf{x}$ are binarized as $\{-1, 1\}$, the above computation reduces to a cardinality constraint of the form $\sum_{i=1}^{m} l_i \geq C$, where $l_i \in \{0, 1\}$ and C is a constant. This cardinality constraint can be encoded as a CNF.

The output block has a LIN layer followed by an ARGMAX layer, which can be encoded using a similar technique. First, we encode a cardinality constraint for all pairs of classes, which tells us the class that has a higher activation function in the pairing. Then, we use a final set of cardinality constraints to determine the class that was the winner in all of its pairings (Narodytska et al., 2018). Since we focus on BNNs with binary output classes in this paper, a single CNF variable is enough to represent the output of the BNN.

The space complexity of this conversion is $O(NC^2)$, where N is the number of neurons in the BNN and C is the constant from the above cardinality constraint.

OBDD to CNF

We convert an OBDD into a CNF using the well-known Tseitin Transformation (Tseitin, 1968), which converts a Boolean circuit into a CNF. Consider an OBDD node labelled by variable X. If the two children of this node compute Boolean functions C_0, C_1, then the OBDD node computes the Boolean function $R = (C_0 \wedge \neg X) \vee (C_1 \wedge X)$. We can then represent the Boolean function of this node by the following five clauses:

$$\begin{aligned} &\neg R \vee C_0 \vee X \\ &\neg R \vee C_1 \vee \neg X \\ &\neg R \vee C_0 \vee C_1 \\ &R \vee \neg C_0 \vee X \\ &R \vee \neg C_1 \vee \neg X \end{aligned}$$

Applying this conversion to all OBDD nodes leads to a CNF representation of the Boolean function computed by the OBDD. The number of CNF clauses produced by this conversion is $5N$, where N is the number of OBDD nodes.

The above encodings allow us to convert a BNN into a CNF α and an OBDD into a CNF β. Let $\mathbf{X}$ be the CNF variables corresponding to the BNN inputs and O be the variable corresponding to its output. Then $\alpha \wedge \mathbf{x} \wedge O$ will be satisfiable iff the BNN outputs 1 under input $\mathbf{x}$. Similarly, $\alpha \wedge \mathbf{x} \wedge \neg O$ will be satisfiable iff the BNN outputs 0 under input $\mathbf{x}$. Now let $\mathbf{X}$ be the CNF variables corresponding to the OBDD variables and let R represent the variable we introduced for the OBDD root. Then $\beta \wedge \mathbf{x} \wedge R$ will be satisfiable iff the OBDD outputs 1 under input $\mathbf{x}$ and $\beta \wedge \mathbf{x} \wedge \neg R$ will be satisfiable iff the OBDD outputs 0 under input $\mathbf{x}$.

When the BNN and the OBDD share the same inputs $\mathbf{x}$, we can check for their inequivalence with the formula $\phi = \alpha \wedge \beta \wedge (O \vee R) \wedge (\neg O \vee \neg R)$ (Narodytska et al., 2018). Then, ϕ is satisfiable iff there is some instantiation of $\mathbf{x}$ such that $(O \wedge \neg R) \vee (\neg O \wedge R)$ (i.e. BNN and OBDD disagree).

4 Exact Learning of Finite Automaton

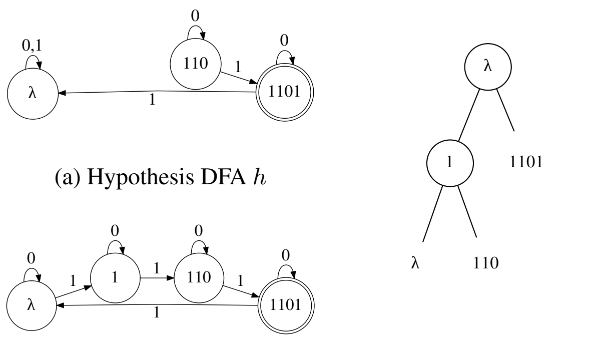
In this section we describe the automaton learning algorithm (Angluin, 1987), which learns Deterministic Finite Automata (DFA). The DFA learning algorithm has an adaptation for learning OBDDs (Nakamura, 2005), which serves as the backbone for our BNN compilation algorithm. DFAs and OBDDs are initimately related, since DFAs are almost the same as Complete OBDDs, which are OBDDs that do not skip variables (Wegener, 2000).

We roughly summarize the exposition on the topic of learning DFAs from the textbook by (Kearns and Vazirani, 1994). The learning algorithm falls under the category of *active* learning where the algorithm can learn through experimentation, as opposed to *passive* learning where the algorithm has no control over the sample of examples. To learn the DFA for a function f, the learning process requires access to oracles for two types of queries:

- Membership Queries: The learning process selects an instance $\mathbf{x}$ and the oracle returns the value of $f(\mathbf{x})$.
- Equivalence Queries: The learner submits a hypothesis automaton h. The oracle tells the learner if h computes the correct function (i.e. $h = f$), otherwise the oracle returns a counterexample $\mathbf{x}$ for which $h(\mathbf{x}) \neq f(\mathbf{x})$.

The main idea of the algorithm is as follows. Let S be the set of states of a minimal DFA we want to learn. Recall that each state represents a distinct equivalence class of input strings. At all times we keep a hypothesis DFA whose states $S^\star$ represent a partition of S. We iteratively refine the partition by splitting some partition element of $S^\star$ into two, so that $|S^\star|$ increases. When $|S^\star| = |S|$, each element in the partition contains exactly one equivalence class from S, so our hypothesis DFA computes the target DFA.

Initially, we start with a one-node hypothesis DFA with just one state, which partitions all the states in S into one group. As long as our DFA is incorrect, we will receive counterexamples from the equivalence query. Given a counterexample e, we can simulate e on our hypothesis DFA



(a) Hypothesis DFA h

(b) Updated DFA

(c) Binary classification tree of h

Figure 2: Learning the finite automaton for the 3 mod 4 counter. Using the counterexample `1101`, we modify the hypothesis DFA into the updated DFA.

and identify the first state $s^\star$ for which its following step in the simulation is provably incorrect. This can be done efficiently by maintaining a binary classification tree, the details of which we omit. Then, we refine the partition by splitting $s^\star$ into two nodes. This process repeats until we have learned all the states of S, at which point the equivalence query gives no more counterexamples and our algorithm terminates.

Suppose we wish to learn a DFA on binary inputs for the 3 mod 4 counter f, and we currently have the hypothesis DFA h in Figure 2a and its binary classification tree in Figure 2c. Since $h(1101) = 0 \neq f(1101)$, we get the string `1101` as a counterexample. Then using the binary classification tree along with membership queries, the algorithm identifies the state λ in h as faulty, and splits it into two. This generates the updated DFA in Figure 2b, which computes f correctly.

The automaton learning algorithm was adapted into an OBDD learning algorithm in (Nakamura, 2005). This variation requires n equivalence queries and $6n^2 + n\log(m)$ membership queries, where n is the number of nodes in the final OBDD and m is the number of variables in the OBDD.

5 BNN Compilation Algorithm

We now describe our main contribution: a compilation algorithm from a BNN to an OBDD. Given a BNN B on n binary inputs and one binary output, we wish to obtain an OBDD D that computes the function of B on a region S (i.e. $D(\mathbf{x}) = B(\mathbf{x}) : \forall \mathbf{x} \in S$). We require region S to be encoded as a CNF.

Algorithm 1 implements our proposal. The subroutines `BNNToCNF` and `OBDDToCNF` perform the encodings described in Section 3. We encode the BNN B as a CNF α with output variable O. Then, we start the OBDD learning algorithm as described in Section 4 to learn the reduced OBDD representation of B. The learning algorithm creates a hypothesis OBDD D, which we encode as a CNF β with variable R representing the OBDD output. We set ϕ on Line 4 such that ϕ has a satisfying assignment iff the current hypothesis OBDD D does not compute the same function as

Algorithm 1 `CompileBNN`$(B, \mathbf{X}, S)$

```
input: A Binarized Neural Network $B$ with input variables $\mathbf{X}$, and a CNF $S$ encoding an input region
output: An OBDD $D$ computing the function of $B$ on $S$
main:
 1: $\alpha, O \leftarrow$ BNNToCNF$(B, \mathbf{X})$
 2: $D \leftarrow$ initial hypothesis OBDD
 3: $\beta, R \leftarrow$ OBDDToCNF$(D, \mathbf{X})$
 4: $\phi \leftarrow \alpha \wedge \beta \wedge (O \vee R) \wedge (\neg O \vee \neg R) \wedge S$
 5: while $\phi$ has a satisfying assignment $\mathbf{s}$ do
 6:     $\mathbf{x} \leftarrow$ projection of $\mathbf{s}$ on $\mathbf{X}$
 7:     $D \leftarrow$ UpdateHypothesis$(D, \mathbf{x})$
 8:     $\beta, R \leftarrow$ OBDDToCNF$(D, \mathbf{X})$
 9:     $\phi \leftarrow \alpha \wedge \beta \wedge (O \vee R) \wedge (\neg O \vee \neg R) \wedge S$
10: return $D$
```

BNN B on region S. While ϕ is satisfiable, we take the satisfying assignment and keep only the variables corresponding to the BNN/OBDD inputs as our counterexample $\mathbf{x}$. The subroutine `UpdateHypothesis` then edits our hypothesis OBDD using counterexample $\mathbf{x}$. Once we have an unsatisfiable ϕ, we return the OBDD D with the guarantee that it computes the same function as BNN B on S. Note that there are no guarantees on the output of OBDD D on instances outside S. The number of iterations of the `while` loop is N, where N is the number of nodes in the final output D.

Algorithm 2 `r-RadiusDomain`$(\mathbf{x}, r)$

```
input: An input $\mathbf{x} = x_1, \ldots, x_n$ and a radius $r \leq n$
output: A CNF that encodes all instances $\mathbf{x}^\star$ such that $h(\mathbf{x}, \mathbf{x}^\star) \leq r$, where $h$ measures the Hamming distance
main:
 1: $d \leftarrow$ a 2D array with dimensions $[0, n] \times [0, r]$
 2: for $j \leftarrow 0$ to $r$ do
 3:     $d_{0,j} \leftarrow \top$
 4: for $i \leftarrow 1$ to $n$ do
 5:     for $j \leftarrow 0$ to $r$ do
 6:         $h \leftarrow d_{i-1,j}$
 7:         $l \leftarrow d_{i-1,j-1}$ if $j > 0$ else $\bot$
 8:         $d_{i,j} \leftarrow$ OBDD node: label $X_i$, $x_i$-child $h$, $\neg x_i$-child $l$
 9: return OBDDToCNF$(d_{n,r}, \mathbf{X})$
```

In Algorithm 2 we propose the construction of an input region that captures all instances in the neighborhood of some instance $\mathbf{x}$ on n variables. More specifically, Algorithm 2 takes in an instance $\mathbf{x}$, a radius r, and outputs a CNF S on variables $X_1, \ldots, X_n$. An instance $\mathbf{x}^\star$ is a satisfying assignment for S iff the Hamming distance between $\mathbf{x}$ and $\mathbf{x}^\star$ is no greater than r. In the algorithm, node $d_{i,j}$ stores the state with $n - i$ variables processed and a current Hamming distance of $r - j$. On Line 8, the child edge of $d_{i,j}$ that agrees with x_i points to $d_{i-1,j}$. The other child edge points to $d_{i-1,j-1}$ if $j > 0$, otherwise it points to $\bot$. By using S as an input for Algorithm 1, we can compile an OBDD that exactly computes the function of a BNN for all instances close to some instance of interest, measured by the number of differing features. The time and space complexity of Algorithm 2 is $O(nr)$.

To extend our algorithm into an anytime compilation algorithm, we start with a small region of interest and increase its size over time. The compiled OBDD D will compute the same function as B on this small region. To compile the OBDD for a larger region, we can feed in D as the initial hypothesis OBDD in Algorithm 1 on Line 2, without the need to build D from scratch. Then, we can use the updated OBDD to verify the properties of B on the enlarged region. We can continue to enlarge this region until $S = \{0,1\}^n$, at which point $S = \top$ and the compiled OBDD computes the same function as B everywhere.

6 Experiments

In this section we present some preliminary experiments as a proof of concept. We use the techniques described in Section 5 to compile the OBDD of a BNN and verify its properties. From the USPS digits dataset, we downsampled and binarized the inputs into 8×8 black and white images (Hull, 1994). We then trained a BNN to distinguish between digit '0' images (true-class) and digit '8' images (false-class), which achieved 94% accuracy using the training algorithm from (Courbariaux et al., 2016). The BNN has 64 input nodes, 5 hidden nodes, and 2 output nodes, and was encoded into a CNF with $10,664$ variables and $41,553$ clauses. Using `riss-coprocessor` to preprocess auxiliary variables, we compressed the CNF to $3,438$ variables and $23,254$ clauses (Kahlert et al., 2015). The original and compressed CNFs are equivalent after existentially quantifying out all variables except for the inputs and output, which is enough for the correctness of our algorithm. Experiments were done using a single Intel(R) Xeon(R) CPU E5-2670 processor.

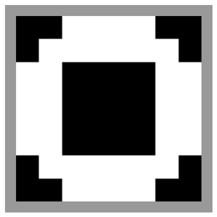
Next, we identified an instance classified as digit '0' (Figure 3a), and compiled the neighborhood around it using Algorithms 1 and 2. We used the `riss` SAT solver for our experiments (Kahlert et al., 2015). Table 2 shows the compilation results for varying values of r. We did the same for an instance that is classified as digit '8' (Figure 3b), and show the compilation results in Table 3. We also compiled around the neighborhood of an image that we believe to be neither a '0' nor an '8' (Figure 3c), and show those results in Table 4. For the experiments with small input space, we verified that the produced OBDD is correct through manual enumeration.

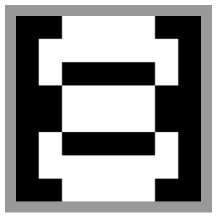
The bottleneck in our experiments is the average time for a SAT query, which is done once for each of the N equivalence queries, where N is the size of the OBDD. We can look more into preprocessing the CNF, since a large portion of the formula is reused for each query. As the OBDD grows, the membership queries become a bottleneck as well since the number of membership queries is quadratic on N.

7 Case Study

In this section we perform verification queries on the setup described in Section 6. We count the number of counterexamples and run prime-implicant queries (PI queries for short), which give us a subset of pixels that render the remaining pixels irrelevant for the BNN classification (Shih, Choi, and Darwiche, 2018b). Let $\mathbf{x}$ be the instance visualized in Figure 3a, classified as a '0' digit. For $r = 5$ in Table 2, the reduced OBDD is just $\top$. This means that flipping any five pixels of $\mathbf{x}$ will still produce another image classified as digit '0.' Recall that an image has 64 pixels in our example, so this classification is robust against changes that modify no more than 7.8% of the image.

For $r = 8$, we get a reduced OBDD of size $5,602$, indicating the existence of counterexamples. Counting the satisfying assignments of an OBDD can be done in time linear in the OBDD size. Utilizing a counting procedure, we found that $227,739,414$ out of the $5,130,659,561$ images (4.4%) are classified incorrectly as the digit '8.' Furthermore, using the PI query, we identified a minimal set of pixels, shown in Figure 4a, that guarantee a correct classification, regardless of how the other pixels are set (within a radius of 8).


(a) A digit 0, classified as '0'.


(b) A digit 8, classified as '8'.

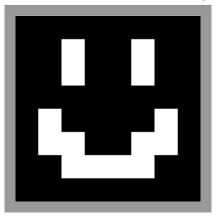
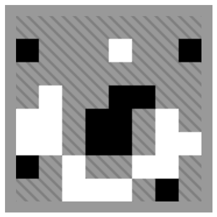

(c) A smile, which happens to be classified as '8'.

Figure 3: Three images: digit 0, digit 8, and a smile. For each image we compile around its r-neighborhood.


(a) 25 out of 64 pixels fixed from Figure 3a

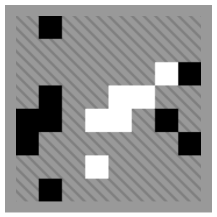
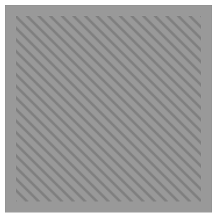
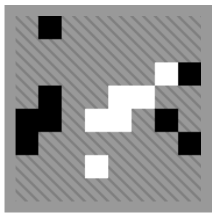

(b) 15 out of 64 pixels fixed from Figure 3b

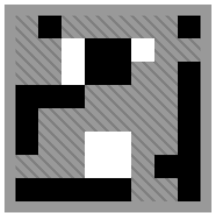
Figure 4: Prime implicant results for the instances shown in Figure 3a and 3b, for $r = 8$. The grey striped region represents the 'don't care' pixels. If we fix the black/white pixels in Figure 4a, any completing image within a radius of 8 from Figure 3a must be classified as '0'. If we fix the black/white pixels in Figure 4b, any completing image within a radius of 8 from Figure 3b must be classified as '8'.



(a) 0 out of 64 pixels fixed from Figure 3a



(b) 14 out of 64 pixels fixed from Figure 3b



(c) 30 out of 64 pixels fixed from Figure 3c

Figure 5: Prime implicant results for $r = 5$ for the instances shown in Figure 3. The grey striped region represents the 'don't care' pixels. If we fix the black/white pixels in Figure 5a, any completing image within a radius of 5 from Figure 3a must be classified as '0'. If we fix the black/white pixels in Figure 5b, any completing image within a radius of 5 from Figure 3b must be classified as '8'. If we fix the black/white pixels in Figure 5c, any completing image within a radius of 5 from Figure 3c must be classified as '8'.

Table 2: Compilation of a BNN on 64 variables around the r-neighborhood of an image of a digit 0 (Figure 3a).

r	input space	OBDD size	compile time (s)
5	8,303,633	0 ($\top$)	2
6	83,278,001	509	403
7	704,494,193	2,202	2,166
8	5,130,659,561	5,602	24,003

Table 3: Compilation of a BNN on 64 variables around the r-neighborhood of an image of a digit 8 (Figure 3b).

r	input space	OBDD size	compile time (s)
4	679,121	0 ($\perp$)	2
5	8,303,633	243	111
6	83,278,001	765	584
7	704,494,193	2,431	3,168
8	5,130,659,561	4,058	11,797

Table 4: Compilation of a BNN on 64 variables around the r-neighborhood of an image of a smile (Figure 3c).

r	input space	OBDD size	compile time (s)
1	65	0 ($\perp$)	1
2	2,081	258	31
3	43,745	1,437	420
4	679,121	6,048	3,336
5	8,303,633	12,297	26,683

We can ask the same queries for the instance $\mathbf{w}$ visualized in Figure 3b and classified as digit '8.' For $r = 4$ in Table 3, the OBDD is just $\perp$, which means that flipping any 4 pixels of $\mathbf{w}$ will still produce another image classified correctly as digit '8.' For $r = 8$, we get an OBDD of size $4,058$. Using this OBDD, we found that $87,761,650$ out of the $5,130,659,561$ images (1.7%) are classified incorrectly as the digit '0.' The PI query identified the minimal set of pixels in Figure 4b which guarantee a correct classification regardless of how the remaining pixels are set (within a radius of 8).

For the "smile" image in Figure 3c, the compiled OBDD for the $(r = 5)$-neighborhood is larger than the corresponding OBDDs of the first two images (see $r = 5$ in Tables 2, 3, 4). As well, for $r = 5$, the PI query for the "smile" requires 30 out of the 64 pixels to be fixed in order to guarantee a classification, while the PI query for the digit '0' and digit '8' only require 0 and 14 pixels respectively (Figure 5). This suggests that the behavior of the BNN is less structured in the region around the image of the "smile", possibly because it is unclear how the image should be classified.

8 Related Work

The success of neural networks has led to a line of work on understanding and verifying their behaviors (Katz et al., 2017; Narodytska et al., 2018; Cheng et al., 2018; Pulina and Tacchella, 2010). These works use, for example, solvers for NP-complete problems such as Mixed-Integer Linear Programming (MILP), satisfiability (SAT), or satisfiability modulo theory (SMT). These systems seek to verify a particular property of a neural network, or otherwise provide a counter-example. We push this line of work further by allowing one to reason about the distribution or the characteristics of counterexamples, which is enabled by learning the OBDD of a given neural network. These richer queries allow us to better understand the neural network behavior beyond detecting the presence of counterexamples.

(Choi et al., 2019) also consider the compilation of neural networks into a tractable representation, and in particular, into a Sentential Decision Diagram (SDD) (Darwiche, 2011; Choi, Xue, and Darwiche, 2012).[1] They focus on a different class of neural networks and take the approach of reducing a neural network to a Boolean circuit, and then compiling the circuit into a tractable one using classical knowledge compilation techniques. While this approach allows a larger set of verification queries, it does not allow local/incremental compilation so it may be less scalable, all else being equal.

Finally, there is also recent work on learning finite state automata from recurrent neural networks (RNNs) (Weiss, Goldberg, and Yahav, 2018; Koul, Fern, and Greydanus, 2019). Some of these works also use an Angluin-style approach for learning the finite state automaton of an RNN (Weiss, Goldberg, and Yahav, 2018). More specifically, their approach is based on learning the finite state automaton of an iteratively-refined abstraction of an RNN's state space, and hence the final automaton learned is not

[1]Note that SDDs are known to be exponentially more succinct than OBDDs (Bova, 2016).

necessarily equivalent to the original RNN. (Koul, Fern, and Greydanus, 2019) train an RNN and then quantize the state space using an autoencoder. The result is a quantized network, whose corresponding state machine can be readily extracted. Angluin-style approaches, including ours, can be viewed as instances of *program synthesis*, where a program (a finite state automaton) is learned from a specification (a neural network). For more on formal synthesis, which lies at the increasingly important intersection of the fields of formal verification and machine learning, see, e.g., (Jha et al., 2017; Jha and Seshia, 2017).

9 Conclusion

We presented new techniques for verifying the behavior of a Binarized Neural Network on some input region. We outlined an algorithm for compiling a BNN into an OBDD on any input region that can be encoded efficiently as a CNF. Our algorithm combines existing methods for CNF encodings with a variation on the classical algorithm for learning DFAs. The compiled OBDD gives us access to a range of efficient verification queries and allows us to reason about counterexamples, such as computing their probability and identifying their common characteristics. In domains such as image classification, our approach can let users pinpoint a specific input image I, and then reason about images that are some pixels away from I but classified differently from I. We showed some preliminary results on a digits classifier, performing verification queries and scaling to 64 inputs and a region of size 5×10^9.

Acknowledgments This work has been partially supported by NSF grant #IIS-1514253, ONR grant #N00014-18-1-2561 and DARPA XAI grant #N66001-17-2-4032.

References

Angluin, D. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75(2):87–106.

Bova, S. 2016. SDDs are exponentially more succinct than OBDDs. In *AAAI*, 929–935.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* C-35:677–691.

Cadoli, M., and Donini, F. M. 1997. A survey on knowledge compilation. *AI Commun.* 10(3-4):137–150.

Cheng, C.; Nührenberg, G.; Huang, C.; and Ruess, H. 2018. Verification of binarized neural networks via inter-neuron factoring (short paper). In *Proceedings of the 10th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, 279–290.

Choi, A.; Shi, W.; Shih, A.; and Darwiche, A. 2019. Compiling neural networks into tractable Boolean circuits. In *AAAI Spring Symposium on Verification of Neural Networks (VNN)*.

Choi, A.; Xue, Y.; and Darwiche, A. 2012. Same-decision probability: A confidence measure for threshold-based decisions. *International Journal of Approximate Reasoning (IJAR)* 53(9):1415–1428.

Courbariaux, M.; Hubara, I.; Soudry, D.; El-Yaniv, R.; and Bengio, Y. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *JAIR* 17:229–264.

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of IJCAI*, 819–826.

Darwiche, A. 2014. Tractable knowledge representation formalisms. In *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press. 141–172.

Hubara, I.; Courbariaux, M.; Soudry, D.; El-Yaniv, R.; and Bengio, Y. 2016. Binarized neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 4107–4115.

Hull, J. J. 1994. A database for handwritten text recognition research. *IEEE Transactions on pattern analysis and machine intelligence* 16(5):550–554.

Jha, S., and Seshia, S. A. 2017. A theory of formal synthesis via inductive learning. *Acta Informatica* 54(7):693–726.

Jha, S.; Raman, V.; Pinto, A.; Sahai, T.; and Francis, M. 2017. On learning sparse Boolean formulae for explaining AI decisions. In *Proceedings of the 9th NASA Formal Methods Symposium (NFM)*, 99–114.

Kahlert, L.; Krüger, F.; Manthey, N.; and Stephan, A. 2015. Riss solver framework v5. 05.

Katz, G.; Barrett, C.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, 97–117. Springer.

Kearns, M., and Vazirani, U. V. 1994. An introduction to computational learning theory.

Koul, A.; Fern, A.; and Greydanus, S. 2019. Learning finite state representations of recurrent policy networks. In *Proceedings of the Seventh International Conference on Learning Representations (ICLR)*.

Leofante, F.; Narodytska, N.; Pulina, L.; and Tacchella, A. 2018. Automated verification of neural networks: Advances, challenges and perspectives. *CoRR* abs/1805.09938.

Meinel, C., and Theobald, T. 1998. *Algorithms and Data Structures in VLSI Design: OBDD — Foundations and Applications*. Springer.

Nakamura, A. 2005. An efficient query learning algorithm for ordered binary decision diagrams. *Information and Computation* 201(2):178–198.

Narodytska, N.; Kasiviswanathan, S. P.; Ryzhyk, L.; Sagiv, M.; and Walsh, T. 2018. Verifying properties of binarized deep neural networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*.

Pulina, L., and Tacchella, A. 2010. An abstraction-refinement approach to verification of artificial neural networks. In *CAV*.

Selman, B., and Kautz, H. A. 1996. Knowledge compilation and theory approximation. *J. ACM* 43(2):193–224.

Shih, A.; Choi, A.; and Darwiche, A. 2018a. Formal verification of Bayesian network classifiers. In *Proceedings of the 9th International Conference on Probabilistic Graphical Models (PGM)*.

Shih, A.; Choi, A.; and Darwiche, A. 2018b. A symbolic approach to explaining Bayesian network classifiers. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*.

Tseitin, G. 1968. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic* 115–125.

Wegener, I. 2000. *Branching Programs and Binary Decision Diagrams*. SIAM.

Weiss, G.; Goldberg, Y.; and Yahav, E. 2018. Extracting automata from recurrent neural networks using queries and counterexamples. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 5244–5253.